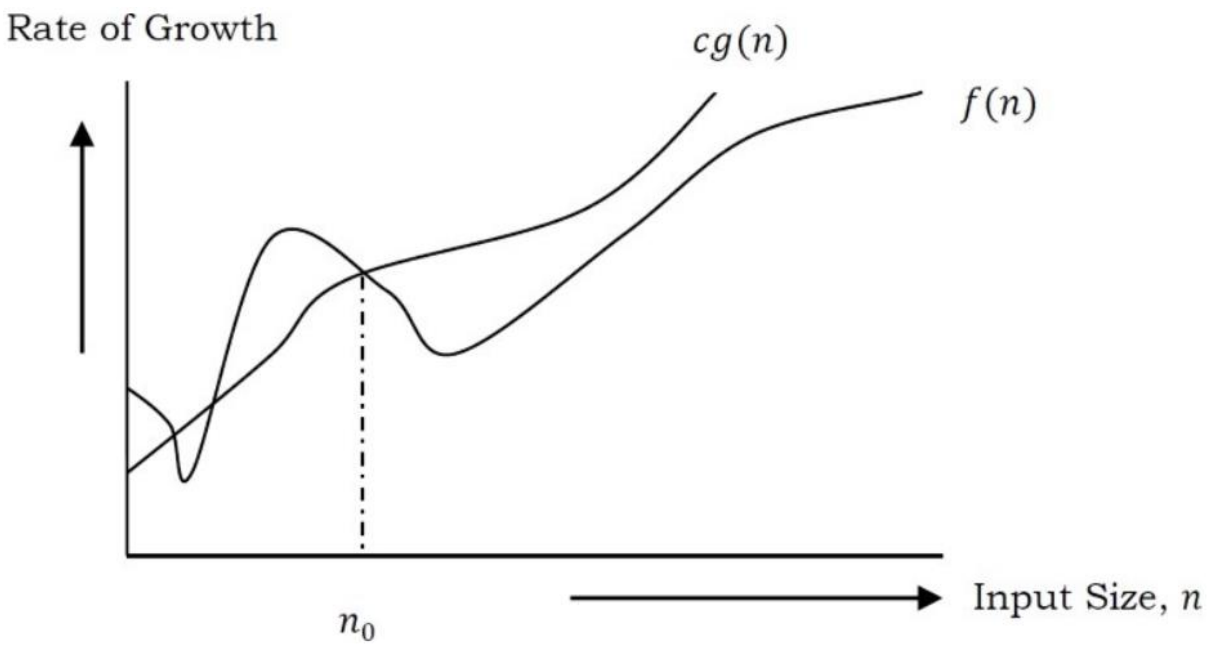
Algorithm Analysis-2

Prof. Dr. Murat GÖK

murat.gok@yalova.edu.tr

Big-O Notation (Upper Bounding Function)

- Big-O notation gives the *tight* upper bound of the given function.
- It is represented as *f*(*n*) = O(*g*(*n*)). That means, at larger values of *n*, the upper bound of *f*(*n*) is *g*(*n*).
    - For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 *is* *g*(*n*). That means *g*(*n*) gives the maximum rate of growth for *f*(*n*) at larger values of *n*.
- O–notation defined as *O(g(n)) = {f(n)*: there exist positive constants c and n_0 such that *0 ≤ f(n) ≤ cg(n)* for all $n > n_0$}
- g(n) is an asymptotic tight upper bound for f(n).
- We discard lower values of *n*. That means the rate of growth at lower values of *n* is not important.
- In the figure, below n_0, the rate of growth could be different. n_0 is called threshold for the given function.



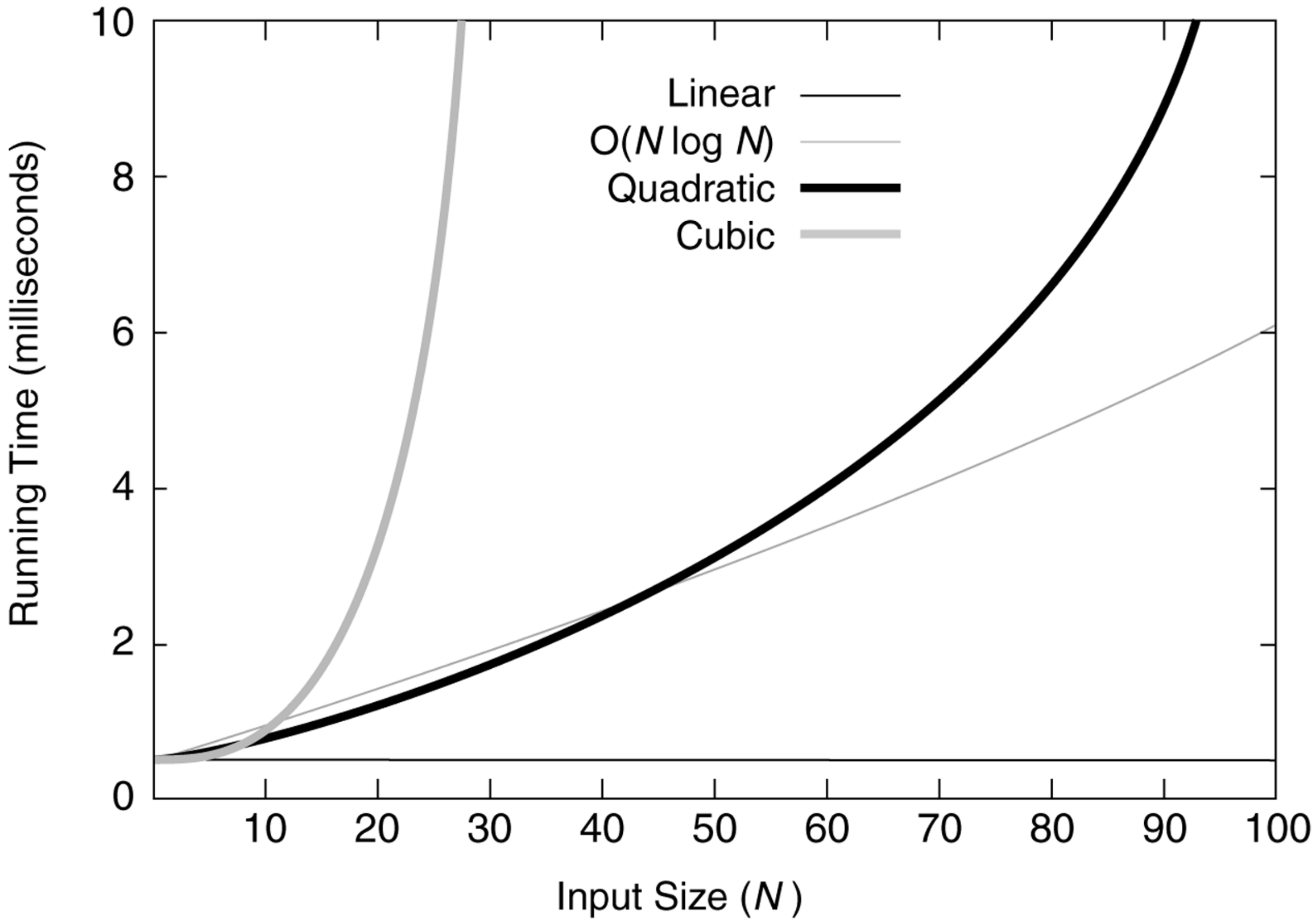


Figure 1

Running times for small inputs

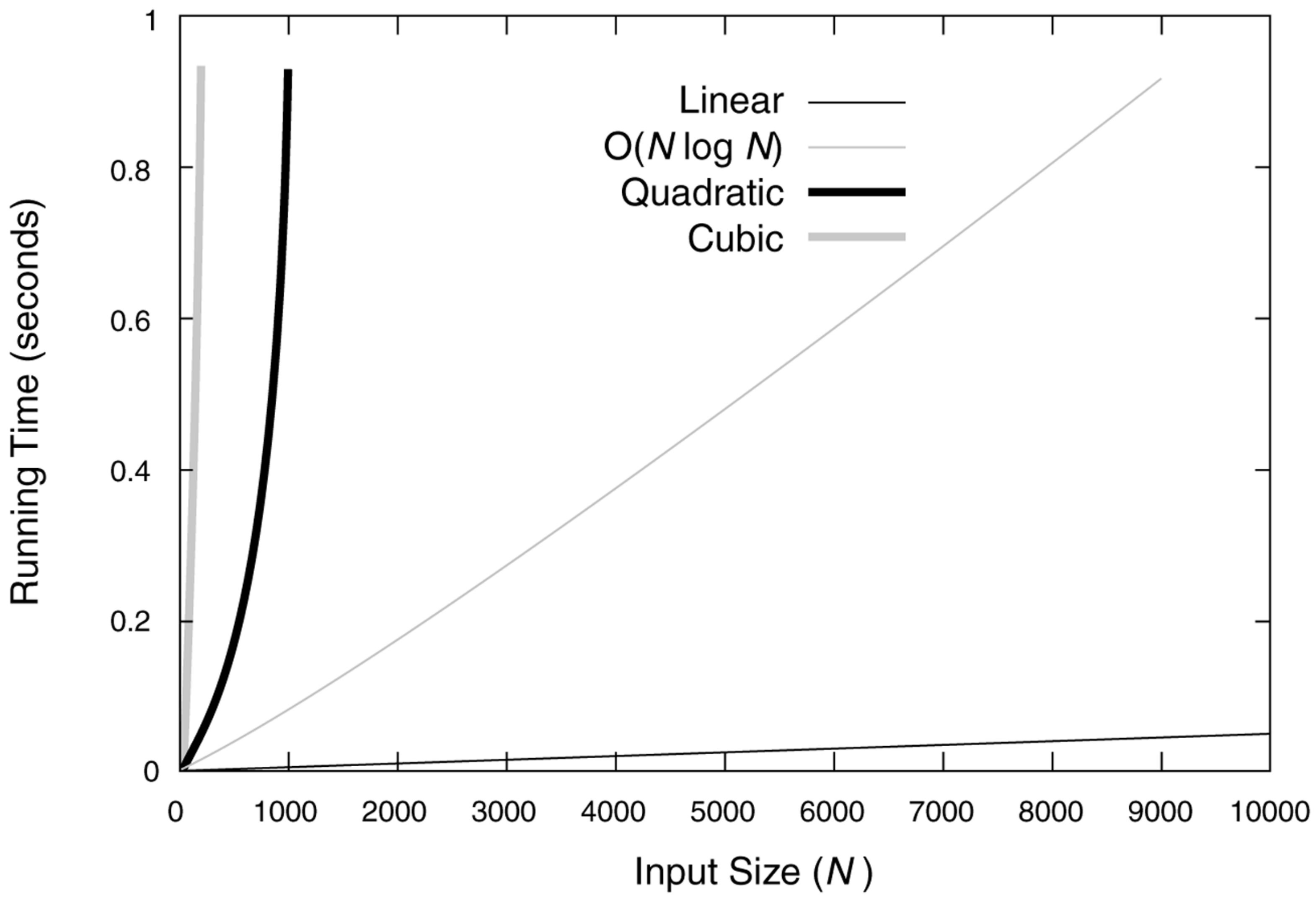


Figure 2

Running times for moderate inputs

Analysis of Algorithms

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

The Execution Time of Algorithms

- Each operation in an algorithm (or a program) has a cost.

  ➔ Each operation takes a certain of time.

`count = count + 1;` ➔ take a certain amount of time, but it is constant

A sequence of operations:

```
count = count + 1;
sum = sum + count;
```

Cost: c_1

Cost: c_2

➔ Total Cost = $c_1 + c_2$

The Execution Time of Algorithms (cont.)

- **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.

```
for (i = 1; <= n; i++) {
m = m + 2;                  // constant time,  c

}
```

Total time = constant c × n = c n = O(n).

➔ The time required for this algorithm is proportional to n

The Execution Time of Algorithms (cont.)

- **Nested Loops**: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.

```
for (i = 1; <= n; i++) {
    // inner loop executes n times
     for (i = 1; <= n; i++) {
          k = k + 1;        // // constant time
}
```

Total time = constant c × n × n = c n^2 = $O(n^2)$.

The Execution Time of Algorithms (cont.)

- **If/Else**: Never more than the running time of the test plus the larger of running times of S1 and S2.

	Cost	**Times**
`if (n < 0)`	c1	1
`absval = -n`	c2	1
`else`		
`absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

The Execution Time of Algorithms (cont.)

Example: Simple Loop

	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

The Execution Time of Algorithms (cont.)

Example: Nested Loop

Code	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to n^2

HW: Investigate the Towers of Hanoi problem in terms of the growth-rate function.

The Execution Time of Algorithms (cont.)

Logarithmic complexity: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
for (i = 1; i<= n;)
      i = i*2;
```

If we observe carefully, the value of *i* is doubling every time. Initially *i* = 1, in next step *i* = 2, and in subsequent steps *i* = 4,8 and so on. Let us assume that the loop is executing some *k* times. At *kth* step 2*k* = *n*, and at (*k* + 1)*th* step we come out of the *loop*. Taking logarithm on both sides (if we assume base-2), gives

$$log(2^{k}) = logn$$
$$klog2 = logn$$
$$k = logn$$

Total time = O(logn).

The Execution Time of Algorithms (cont.)

- Similarly, for the case below, the worst case rate of growth is O(logn). The same discussion holds good for the decreasing sequence as well

```
for (i = 1; i<= n;)
     i = i/2;
```

- Another example: binary search (finding a word in a dictionary of *n* pages)
  - Look at the center point in the dictionary
  - Is the word towards the left or right of center?
  - Repeat the process with the left or right part of the dictionary until the word is found.

Some Growth-rate Functions

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + \ldots + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$$

Example

- What is the complexity of the program given below:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j= j+1)
                        for(k=1; k<=n; k= k * 2)
                                count++;
```

Example

- What is the complexity of the program given below:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```


Thank you for
your attention